

---

# Aspect Oriented Programming

Paolo

*May 10, 2002*

# Why AOP?

---

**Robert Martin (SD West April 2002)**

*Forget about the new paradigm, OOP is mainly a tool to manage dependencies in large applications.*

*But it has its limitations: while OOP is great to handle new types added to an existing application, procedural programming is often superior when it comes to extend the functionality provided by the application.*

# What is AOP?

---

**At the design level, an Aspect is a concern that crosscuts class and components hierarchies**

**At the programming level an Aspect is a construct that allow to modularize a crosscutting concern**

**The most popular AOP language is Xerox PARC's aspectJ ([www.aspectj.org](http://www.aspectj.org)), which has spawned aspectC++ ([www.aspectC.org](http://www.aspectC.org)) aspectC ([www.cs.ubc.ca/labs/spl/projects/aspectc.html](http://www.cs.ubc.ca/labs/spl/projects/aspectc.html))**

# One Interface Many Implementations

---

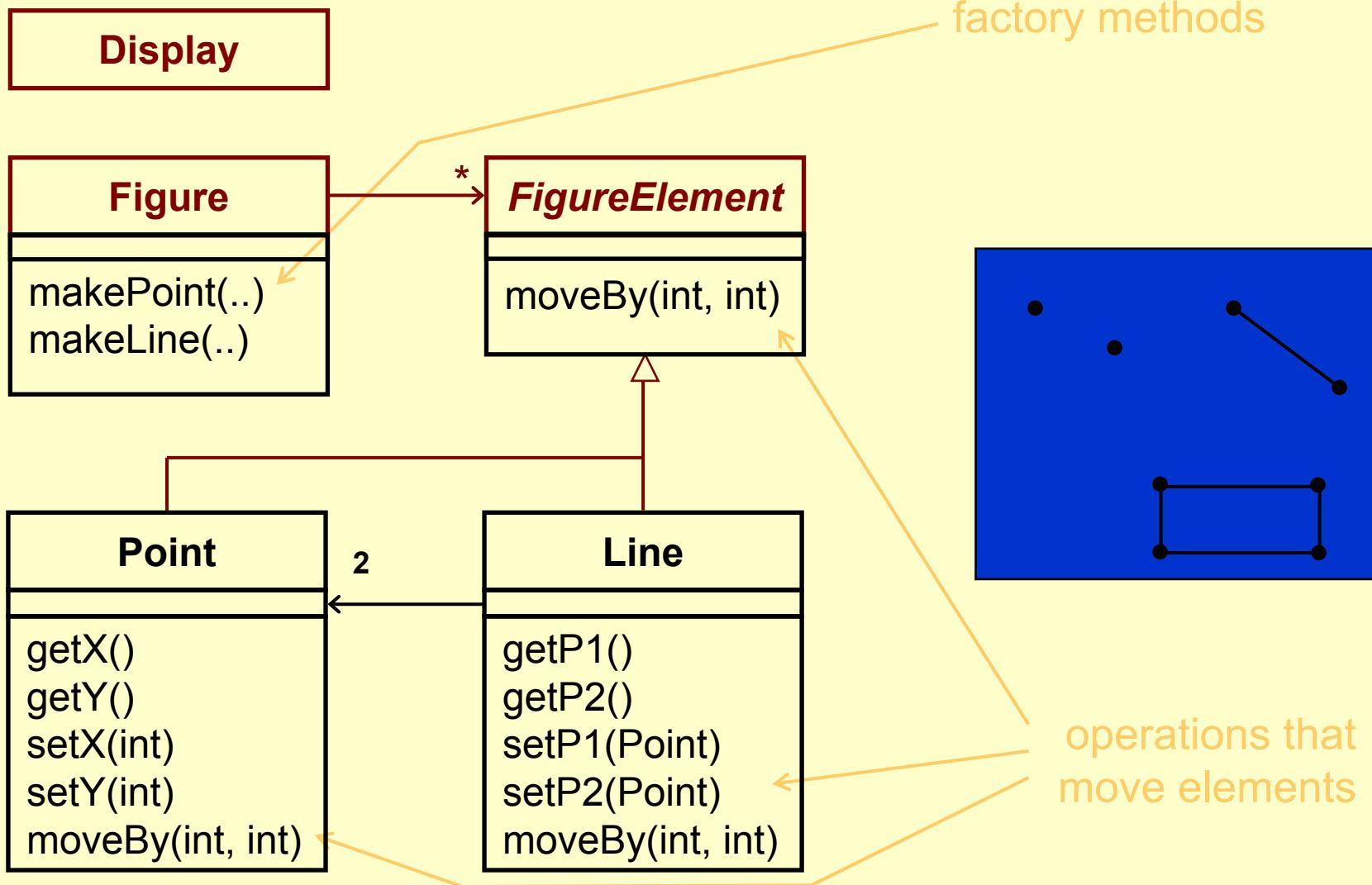
OOP promotes separation of concerns, each object is in charge of implementing the interface it declares.

Common concerns (logging, testing, caching, persistency) end up scattered across many classes and often tangled to each other.

Inheritance and delegation alleviate the problem of cut-and-paste code but create top-heavy code that is difficult to maintain

Generative programming techniques are a better tool to add functionality but sometimes (esp in C++) are too complex

# Figure Editor



# Original Code

---

```
class Line {  
    private Point p1, p2;  
  
    Point getP1() { return p1; }  
    Point getP2() { return p2; }  
  
    void setP1(Point p1) {  
        this.p1 = p1;  
    }  
    void setP2(Point p2) {  
        this.p2 = p2;  
    }  
}  
  
class Point {  
    private int x = 0, y = 0;  
  
    int getX() { return x; }  
    int getY() { return y; }  
  
    void setX(int x) {  
        this.x = x;  
    }  
    void setY(int y) {  
        this.y = y;  
    }  
}
```

# Modify to update display

---

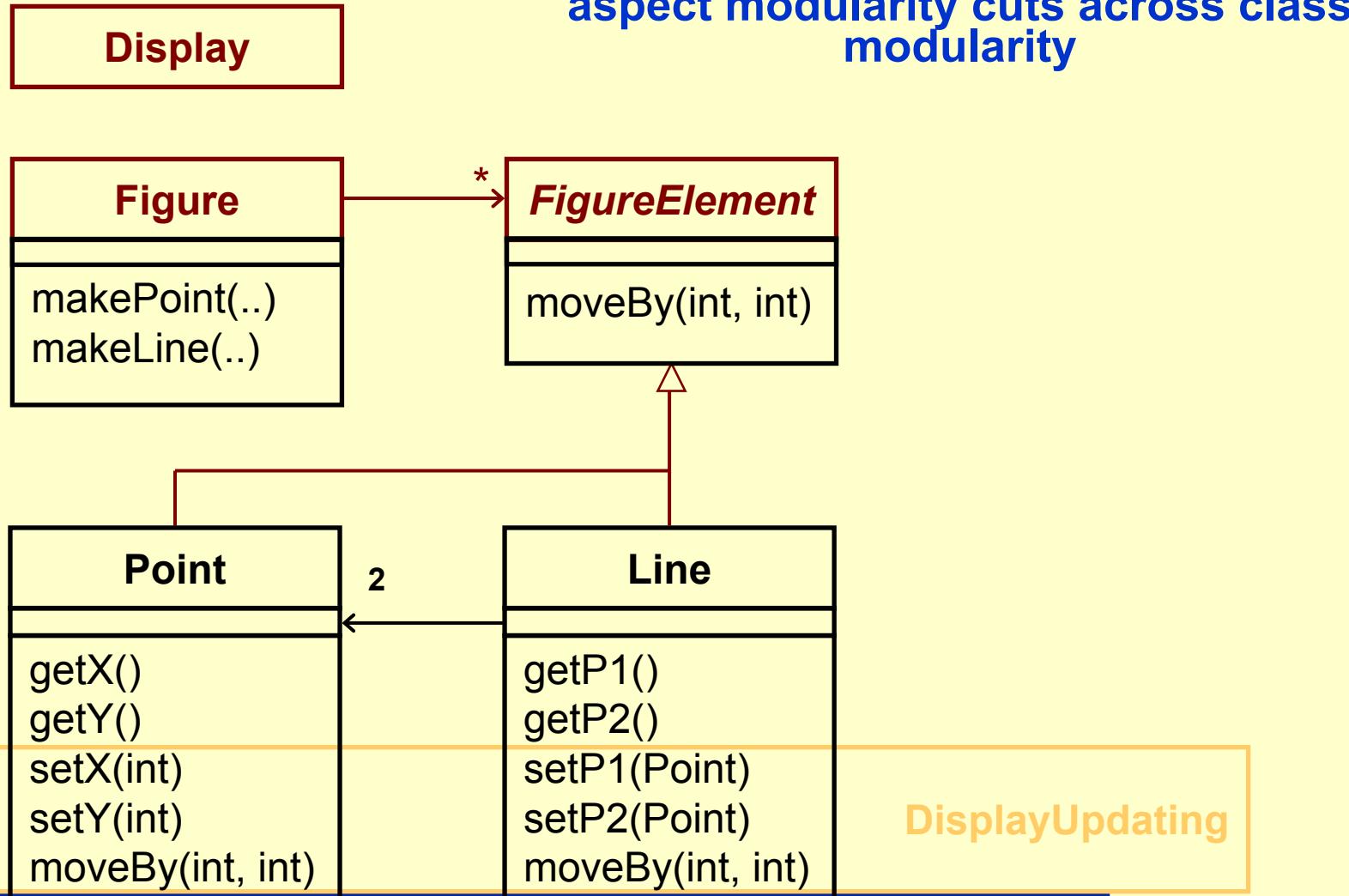
```
class Line {  
    private Point p1, p2;  
  
    Point getP1() { return p1; }  
    Point getP2() { return p2; }  
  
    void setP1(Point p1) {  
        this.p1 = p1;  
        Display.update(this);  
    }  
    void setP2(Point p2) {  
        this.p2 = p2;  
        Display.update(this);  
    }  
}  
  
class Point {  
    private int x = 0, y = 0;  
  
    int getX() { return x; }  
    int getY() { return y; }  
  
    void setX(int x) {  
        this.x = x;  
        Display.update(this);  
    }  
    void setY(int y) {  
        this.y = y;  
        Display.update(this);  
    }  
}
```

## no locus of “display updating”

- evolution is cumbersome
- changes in all classes
- have to track & change all callers

# Update Aspect crosscut classes

---



# Again the Original Code

---

```
class Line {  
    private Point p1, p2;  
  
    Point getP1() { return p1; }  
    Point getP2() { return p2; }  
  
    void setP1(Point p1) {  
        this.p1 = p1;  
    }  
    void setP2(Point p2) {  
        this.p2 = p2;  
    }  
}  
  
class Point {  
    private int x = 0, y = 0;  
  
    int getX() { return x; }  
    int getY() { return y; }  
  
    void setX(int x) {  
        this.x = x;  
    }  
    void setY(int y) {  
        this.y = y;  
    }  
}
```

# Modified using aspectJ

---

```
class Line {  
    private Point p1, p2;  
  
    Point getP1() { return p1; }  
    Point getP2() { return p2; }  
  
    void setP1(Point p1) {  
        this.p1 = p1;  
    }  
    void setP2(Point p2) {  
        this.p2 = p2;  
    }  
}  
  
class Point {  
    private int x = 0, y = 0;  
  
    int getX() { return x; }  
    int getY() { return y; }  
  
    void setX(int x) {  
        this.x = x;  
    }  
    void setY(int y) {  
        this.y = y;  
    }  
}
```

```
aspect DisplayUpdating {  
  
    pointcut move(FigureElement figElt):  
        target(figElt) &&  
        (call(void FigureElement.moveBy(int, int)) ||  
         call(void Line.setP1(Point)) ||  
         call(void Line.setP2(Point)) ||  
         call(void Point.setX(int)) ||  
         call(void Point.setY(int)));  
  
    after(FigureElement fe) returning: move(fe) {  
        Display.update(fe);  
    }  
}
```

## clear display updating module

- ❑ all changes in single aspect
- ❑ evolution is modular

# An Aspect in aspectJ

---

```
aspect DisplayUpdating {  
  
    pointcut move(FigureElement figElt):  
        target(figElt) &&  
        (call(void FigureElement.moveBy(int, int)) ||  
         call(void Line.setP1(Point)) ||  
         call(void Line.setP2(Point)) ||  
         call(void Point.setX(int)) ||  
         call(void Point.setY(int)));  
  
    after(FigureElement fe): move(fe) {  
        Display.update(fe);  
    }  
}
```

**Join Points**

**Advice**

# aspectJ Join Points Designators

---

**when a particular method body executes**

- execution(void Point.setX(int))

**when a method is called**

- call(void Point.setX(int))

**when an exception handler executes**

- handler(ArrayOutOfBoundsException)

**when the object currently executing is of type SomeType**

- this(SomeType)

**when the target object is of type SomeType**

- target(SomeType)

**when the executing code belongs to class MyClass**

- within(MyClass)

**in the control flow of a call to Test's main() method**

- cflow(void Test.main())

**Logical operations, WildCarding and Composition available:**

- within(\*) && execution(\*.new(..))
- execution(public !static \* \*(..))
- cflow(fooPCut() && barPCut())

# aspectJ advices

---

## before advice

- ❑ Runs before entering the join point

```
before(FigureElement fe) : move(fe)  
{ System.out.println("About to move figure " + fe); }
```

## after advice

- ❑ Runs on the way back out

```
after(FigureElement fe): move(fe) { Display.update(fe); }
```

- ❑ “after returning” advice (gives access to the return value)
- ❑ “after exception” advice (gives access to the exception)

## around advice

- ❑ Runs *instead* of the join point. The original join point action can be invoked via the *proceed* call.
  - check pre-/post-conditions, update cache, resource cleanup...

# Development Aspects

---

## Tracing

## Check Invariants

### ❑ Compile-time

- Warn if a non-public field is set outside a setter method
- Warn if an object is constructed outside a factory method
- Error if a destructor throws
- Error if deleting a const pointer

### ❑ Run-time

- Verify/enforce pre- post- conditions

## Logging/Testing

### ❑ Memory/resource usage (cfr Athena Auditors)

### ❑ Record input args and global context on error/exception

### ❑ Replay a suite of test values to a group of methods

# Tracing

---

```
cd /home/calaf/java/aspectj/examples/  
ajc -argfile spacewar/demo.lst  
java spacewar.Game  
ajc -argfile spacewar/debug.lst  
java spacewar.Game
```

# Checking invariants

---

- Contract enforcement

```
static aspect RegistrationProtection {  
    pointcut register(): call(void Registry.register(FigureElement));  
    pointcut canRegister(): withincode( * FigureElement.make*(..));  
    declare error: register() && !canRegister() :  
        "only factory methods can register new figure elements"  
}
```

- Pre- post- conditions (a la Eiffel)

```
aspect PointBoundsChecking {  
    private boolean Point.assertX(int x) {return (x <= 100 && x >= 0);}  
    before(Point p, int x): call(boolean p.setX(x))  
    { if (!p.assertX(x)) {throw InvalidArgumentException(x); }  
    after(Point p, int x) returning (boolean err): call(boolean p.setX(x))  
    { if (err) { throw Exception("setX(\"+x\") failed for " + p); }  
    ...  
}
```

# Production Aspects

---

## Change Monitoring

- ❑ The Display.Update example

## Context Passing

- ❑ Avoid passing args down deep calling sequences

## Providing Consistent Behaviour

- ❑ E.g. for exception handling

## Caching

- ❑ Calculation results or lazy instantiation

# Context Passing

---

- Letting a called method know the identity of the caller

```
Aspect CallerID {  
    pointcut record(StoreGateSvc called):  
        target(called) &&  
        execution(StatusCode StoreGateSvc.record(..));  
    pointcut algexec(Algorithm caller):  
        this(caller) &&  
        cflow( StatusCode *.execute());  
    before(StoreGateSvc called, Algorithm caller):  
        record(StoreGateSvc called) &&  
        algexec(Algorithm caller)  
        { called.setCurrentAlg(caller); }  
}
```

# Introductions

---

An introduction is an aspect member that allows to

- add methods to an existing class
- add fields to an existing class
- extend an existing class with another
- implement an interface in an existing class
- convert checked exceptions into unchecked exceptions

```
aspect CloneablePoint {  
    declare parents: Point implements Cloneable;  
    declare soft: CloneNotSupportedException:  
        execution(Object clone());  
    Object Point.clone() { return super.clone(); }  
}
```

# Caching (in aspectC++)

---

## aspectC++ syntax quite similar

```
aspect Cache {  
    advice call("% square(...)") : void around () {  
        if (!cache.find(*(unsigned*)thisJoinPoint->arg(0),  
                        (unsigned*)thisJoinPoint->result())) {  
            thisJoinPoint->action().trigger();  
            cache.insert(*(unsigned*)thisJoinPoint->arg(0),  
                        *(unsigned*)thisJoinPoint->result());  
        }  
    }  
};
```

## aspectC++ is a code generator (while aspectJ is a real compiler)

- Quite rough compared to aspectJ (no docs, poor build support), but the examples work as far as I can tell

# Conclusion

---

I am impressed!